OS - Slides summary

Operating systems (Technische Universiteit Eindhoven)

Scan to open on Studeersnel

Studocu is not sponsored or endorsed by any college or university
Downloaded by Anonymous Student (danieltyukov1@gmail.com)

OS: place in the computer system

Elements of a computer system:

- Users: humans and machines
- Software: application programs, OS, system programs
- Hardware: CPU, memory (storage), battery, input/output devices

Hardware operates at a very low abstraction level compared to user programs. A mediator in between is needed.

Hardware resources are shared among running programs.

Also, hardware can be expensive. Therefore, the use of HW resources needs to be optimized.

Informal definition operating system:

- A piece of software that acts as intermediary between users/applications and computer hardware
- Provides a level of abstraction that hides the gory details of the hardware architecture from the applications.
- Organizes sharing of HW resources among applications and users (virtualization)
- Maximizes computer performance (resource management)

OS kernel: the one program that always runs. Everything else is either a system program or an application program.

How a computer works:

CPU and I/O devices execute concurrently. I/O: data transfer between device and the local buffer or device controller. In the meantime, CPU does other things.

Device controller (Hardware) informs CPU that the device is ready for I/O by means of an interrupt.

CPU moves data between Main Memory and these local buffers.

Alternative: Direct Memory Access (DMA)

OS's are interrupt driven.

An **interrupt** is a signal to the CPU that transfers control to an interrupt service routine. It is caused by hardware. The *interrupt vector* contains addresses of all service routines. The OS preserves CPU state by storing registers and program counter (address of the interrupted instruction is saved)

A **trap** or **exception** is a software-generated interrupt. It is caused either by a software error or by a system call (explicit SuperVisor Call or SVC), in line with control flow of program.

OS: Dual-mode operation.

Dit document is beschikbaar op studeersnel

Downloaded by Anonymous Student (danieltyukov1@gmail.com)

Different privileges required for different types of code. Most instructions (user code) can be executed in user mode. Some “privileged” instructions are only executable in kernel mode.

Dual-mode allows OS to protect itself and other system components. **Mode bit** is provided by the hardware. This provides the ability to distinguish when the system is running user code or kernel code. System call changes mode to kernel, return from call resets it to user.

status = read(fd, buffer, nbytes);

the filepointer, fd, refers to a data structure (probably within the kernel space) that stores information where the file is to be found, current access state (particularly, the read position) etcetera.

The requested data can be found in system buffers that store disk blocks. Already available or needing disk access.

Copying data from disk to these buffers is by hardware read requests that specify disk block and destination buffer and issue the disk operation and wait for completion interrupt.

The file system may do look-aheads concurrently with user activities.

When this data is not available in system buffers at the time of reading, this results in the suspension of the calling process. Similar suspension occurs when the process has consumed its allotted time.

Some common notions:

Policy: defines *what* you want a system to do

Mechanism: defines how to do it.

Policies need appropriate mechanisms for their realization.

Transparency: hide details with respect to a given issue.

Virtualization: provides a simple, abstract, logical model of the system. Current systems virtualize the entire hardware.

Layering and virtualization:

An OS layer with a well-defined API yields a virtual machine. For a programmer, the virtual machine is what he sees. Porting programs amounts to porting such virtual machines.

Virtualization can support several OS’s on top of one OS. This is implemented on a run-time virtualization layer.

This virtualization can go down to immediately above the hardware.

Motivation and OS tasks:

OS motivation: deal with diversity. CPU and main memory are required for basic operation. There are many alternatives.

Diverse CPUs: instruction set architectures

Diverse architectures: multi-processor with shared memory, multi computer with private memory and shared file system, independent computers on a network

Downloaded by Anonymous Student (danieltyukov1@gmail.com)

OS motivation: transparency and virtualization. OS task: for example hide inner workings and details of processing platform, or abstract from the complicated memory hierarchy and physical limitations.

OS Motivation: Shared functionality.

Large collection of services needed by virtually all programs.

User interface, program execution, I/O operations, file-system management, communication between processes, error detection, resource sharing and allocation, accounting, protection, security.

OS task: provide functionality common to most programs. Introduce well-defined abstractions of concepts. Provide system calls for functions provided by the OS.

System calls: programming interface to the services provided by the OS. Typically written in a high-level language.

System calls are mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use.

API's are needed because 1) direct system calls are more difficult to work with than an API, 2) system calls can be very frequent and 3) a program written using an API can compile/run on any system supporting the API.

Types of system calls: process management, file management, memory management, device management, communications, miscellaneous.

OS motivation: concurrency.

The machine must be shared: multiple activities ('tasks', 'processes') and multiple users. Devices and processor(s) operate concurrently and many common OS tasks are always running; termination is rare.

OS task: realize concurrency transparency (virtualization) and manage and protect resource usage limits between tasks and between users.

OS motivation: portability.

Protect investments in application software. Support source code portability.

OS task: give a unified machine view to applications. Standardize on the API – effectively, define a virtual machine.

Extra-functional requirements:

- Efficiency: the sacrificed efficiency of having an OS rather than direct access should be reasonable. Rule of thumb: If a function can be implemented with the available ones, don't provide it.
- Extensibility: support for adding application-specific functionality
- Scalability: wide range of environments, functionalities, machines
- Dependability: robust, correct, safe and secure.

Real- time OS:

Dit document is beschikbaar op studeersnel

Downloaded by Anonymous Student (danieltyukov1@gmail.com)

- Predictability
- Support for dealing with real-time control
- Stringent dependability

Embedded OS

- Small footprint
- Low system requirements
- Stringent dependability

Processes:

Process concept: a process is a program in execution. OS processes run in kernel mode, user processes run in user mode.

A process includes: program code, stack, heap, data, program counter.

Memory allocated to a process:

Text (program code), stack, heap and data are memory segments allocated to a process.

Stack (temporary data): local variables, function parameters, return addresses etcetera. Size known at compile time.

Heap (dynamically allocated during runtime): used to store data whose size may be unknown before runtime

Data (global variables): fixed size.

Process Control Block (PCB) inside Kernel

A process is identified by an ID and a pointer to a PCB in kernel space. PCB contains: process state, process number, program counter (address of next instruction), CPU registers, Memory management info, I/O status information, scheduling information, accounting information and more.

Process states:

New: process is being created

Ready: process is waiting to be assigned to a CPU (before scheduler dispatch)

Running: Instructions are being executed

Waiting: Process is waiting for some event to occur

Terminated: Process has finished execution

Process scheduling

Ready queue and device (I/O) queues. Queues are generally stored as a linked list.

Context switch

Context switch: saving the state of a process whose execution is to be suspended, and reloading this state when the process is to be resumed.

Downloaded by Anonymous Student (danieltyukov1@gmail.com)

Overhead: (typically) takes a few msecs. Depends on hardware.

How to create/terminate processes

Process creation

Starting point is an already running process that was started through a login procedure

Parent process creates children processes, which, in turn, create other processes, forming a tree of processes. New processes are given process identifiers (pid)

Resource sharing options:

- Parent and children share all resources.
- Children share a subset of the parent's resources.
- Parent and child share no resources.

Execution options

- Parent and children execute concurrently
- Parent waits until children terminate

Address space options

- Child is a duplicate of parent
- Child has another program loaded into its address space

Portable Operating System Interface (POSIX) API for process creation / termination

IEEE standard on the API (it's not an OS). Goal: reduce portability effort for applications. Many operating systems use POSIX API.

A system supporting POSIX provides: a host language and compiler, programming interface definition files, programming interface implementation binary or code, a run-time system.

POSIX processes

POSIX 1003.1 API: fork() – exec() – exit() – wait()

fork() system call creates new process (duplicate address space of the parent in another location)

exec() system call used after a fork to write over the process memory space with a new program

wait() system call may be issued by the parent to wait until the execution of the child is finished.

POSIX/UNIX processes in operation (Solaris example)

Unique pid's

Parent-child relationships are recorded

System services are processes started by init (pid =1), by themselves capable of starting new ones.

Init initiates dtlogin for login of new users

Dit document is beschikbaar op studeersnel

Downloaded by Anonymous Student (danieltyukov1@gmail.com)

Init initiates inetd (the internet daemon) for network services.

Example

execlp() overwrites the calling process with its argument (an executable). This means that code following execlp is never reached.

perror() is a generic error printing routine.

Process termination

Process may ask the operating system to delete itself (exit).

-return status value from child to parent (via wait)

-process' recourses are taken away by OS.

Parent may terminate execution of children processes (abort).

- Child has exceeded allocated resources
- Task assigned to child is no longer required
- If parent is exiting (some operating systems do not allow child to continue if its parent terminates, in which case all children are terminated. Some operating systems attach orphans to a 'grandfather' process)

Termination of children

Use exit(status) to terminate

In many systems, parent needs to wait for children to free child's resources.

Functions wait(), waitpid() → wait() blocks until a child returns

Alternative: use asynchronous notification signals.

Process communication

Cooperating processes need interprocess communication (IPC)

- No shared address space: useful for exchanging small amounts of data
- Shared address space: faster but difficult to implement

Shared memory between processes

A memory segment is reserved in the kernel memory. It is added to the space addressable in user mode. Since it is in the shared space, it is available to other processes as well. No kernel involvement is required upon subsequent reading and writing.

Major issue: interference. There is mutual influence (good or bad):

- Assumptions or knowledge that one process has about the state, are disturbed by actions of another process.

Threads

Process vs thread

Downloaded by Anonymous Student (danieltyukov1@gmail.com)

Disadvantages of process parallelism:

- Shared memory allocation overhead: OS involvement. Inter-process communication (IPC) is expensive.
- Creation overhead (much higher than creating a thread)
- Process switching overhead: mode + context switch. For each IPC on a single-processor system. Save/restore state.

Process vs Thread:

Process:

- Unit of work
- Defines a memory space
- Defines ownership of other resources
- Has at least one associated thread of execution

Thread:

- Unit of concurrency and scheduling
- All threads of a process can operate in the process' address space
- Has an associated execution state (PCB is extended with thread info)

Multithreading

Structuring by threads

The program must deal with three input sources.

Inefficient program structure (busy-waiting):

While … do

Checkinput1;

Checkinput2;

Checkinput3;

Wait a while

Od

Simpler structure:

- Start three threads
- Each thread takes care of one source (and they use interrupts)

Reasons to introduce threads

- Increase concurrency level (less costly). Performance: hide latency (while waiting CPU can do something useful in another thread), increase responsiveness, exploit platform concurrency

Dit document is beschikbaar op studeersnel

Downloaded by Anonymous Student (danieltyukov1@gmail.com)

(multiple processors → multiple threads). Discriminate importance levels in activities (e.g. interrupt routines)

- "Natural" concurrency. Natural organization, structure, e.g. thread per event, thread per resource, thread per (active) external interaction sequence.

In multithreaded processes process address space and resources are shared.

Concurrent execution: single-core vs multi-core system

Considerations

- Dividing the program into activities that can be executed in parallel
- Load balancing
- Interference issues (solved by synchronization)

Summary

- Concurrency on memory address space of the process . No protection against other threads (faults, memory sharing). Inexpensive communication between threads.
- State per thread. Thread context: CPU registers, stack image.
- Generally, little overhead for switching. Depending on the underlying thread execution model

Multithreading models

User and kernel threads

Threads are supported either at user level or by kernel.

User threads are supported above the kernel (no kernel support) via thread libraries. Three primary thread libraries: POSIX Pthreads, Win32 threads, Java threads.

Kernel threads are supported by all modern OS (kernel support). Windows, solaris, Linux, unix (some versions), Max OS X.

Thread libraries can also be used on top of OS's supporting kernel threads. Needed: a mapping model between user threads and kernel threads.

Multithreading mapping models

-many-to-one

-one-to-one

-many-to-many (plus a variation: two-level mode)_

Many-to-one model

User threads mapped to a single kernel thread.

Advantage: thread management by thread library. No need for kernel involvement, making it fast.

Downloaded by Anonymous Student (danieltyukov1@gmail.com)

Disadvantage: only one user thread can access kernel at a given time. Multiple threads cannot run concurrently on different processors. A blocking system call from one user thread blocks all user threads of the process.

One-to-one model

Each user-level thread maps to kernel thread.

Low performance in case of many user threads.

Examples: windows, linux, solaris 9 and later.

Many-to-many model

Allows OS to create a (limited) number of kernel threads. #kernel_threads <= #user_threads.

Examples: windows NT/2000 with the threadfiber package, solaris 8 and earlier.

Concurrency level is limited by the number of kernel threads.

Two-level model

Similar to many-to-many, except that it allows a user thread to be bound to a kernel thread.

Thread execution

Issue: mapping between kernel and user threads

Lightweight process (LWP) as a user-level representation of a kernel thread. It's an intermediate data structure, like a virtual processor for user threads and threadlib.

Kernel schedules LWPs. User threads are scheduled onto an available LWP. Blocking calls in user space or call to thread library package switch LWP to a new user thread. A user thread can also be bound to a LWP.

Concurrency level: number of kernel threads.

Thread switching

Threads are executed in an interleaved way. Switching between threads: two possibilities:

- Switching as a kernel activity. Kernel maintains execution state of thread. Similar to process switching, but without context.
- Switching handled by user level thread package (library). The package maintains execution state of thread, the package must obtain control in order to switch threads. Responsibility of programmer to call package – 'cooperative multithreading'. Could use timer interrupt.

Thread pools

- Create a number of threads in a pool where they await work.

Advantages:

- Usually slightly faster to service a request with an existing thread than creating a new thread
- Allows the number of threads in the application(s) to be bound by the size of the pool

Pthread interface, in C

Dit document is beschikbaar op studeersnel

Downloaded by Anonymous Student (danieltyukov1@gmail.com)

Start function as a new thread (setting attributes, e.g. stacksize;)

This thread terminates when this function returns

Extensive interface (detach: clean up when thread terminates, join: pick up terminated thread, cancel: stop a thread)

Scheduling

Resource scheduling: policies/mechanisms

A single process (thread) may not be able to utilize system resources.

Example: single CPU.

Process with alternating sequence of CPU and I/O bursts. Burst lengths are of importance. CPU bound vs I/O bound programs.

Only one process can run on a processor at any time instant! Scheduling is needed.

Resource scheduling (allocation)

Assignment of resources to tasks. Schedule S is a function that maps a time and a resource to a task. S(t,r) = P means that task P is assigned resource r at time t.

Def'n of a task is context dependent. E.g. a process, a thread of execution, the reading of a disk block, handling an interrupt etcetera.

The changes in a schedule are interesting to examine.

- When to switch to a decision mode for e.g. a processor or a memory segment?

  Processor: e.g. a process into ready queue, end of time slice, process yielding

  Memory: e.g. memory management call; replacement policy by memory subsystem, process termination.

- Decision procedures are used when in decision mode

Schedulilng policies and mechanisms

Scheduling policy represents the strategy for allocating a resource to a task while in decision mode.

Policy: an algorithm that decides based on scheduling criteria or based on a pre-computed lookup table.

Scheduling (allocation) mechanisms are different per resource type.

Processor: implicit management at scheduling points by OS (not visible in program text), through saving and restoring context.

Passive resource: explicit locking and unlocking in program text (e.g. using semaphores), implicit management through the OS, e.g. for memory

CPU scheduling

Metrics for the quality of CPU scheduling

Scheduling Criteria

Downloaded by Anonymous Student (danieltyukov1@gmail.com)

Metric: observed property, result of applying scheduling policy

CPU utilization: keep the CPU as busy as possible

Throughput: # of processes that complete their execution per time unit

Turnaround time: time to execute a particular process

Waiting time: time a process has been waiting in the ready queue

Response time: time it takes from when a request was submitted until the first response is produced -> not the entire output (suitable for time-sharing interactive environment)

- Most texts: response time is defined as the time elapsing from arrival to completion.

Number of deadline misses (real-time scheduling)

Time attributes of a task

A task has

- A name (the j^{th} task) t_j
- A (worst case) execution time S_j
- A period, sometimes T_j
- A relative deadline, sometimes D_j

Dynamic time attributes (i^{th} instance or occurrence)

- An arrival time $a_{j,i}$
- An absolute deadline, sometimes (add D_j to arrival time) $dl_{j,i}$
- A start time (or beginning time) of execution $b_{j,i}$
- A departure time, or end time $e_{j,i}$

Response time: beginning time – arrival time

Turnaround time: end time – arrival time.

CPU scheduling

Scheduling framework

- Decision mode: when to schedule
- Priority function (priority scheduling): who to schedule based on priority
- Arbitration rule: who to schedule in case of equal priority

Managed by 2 different OS modules (jointly called the scheduler *ominous music*)

Process scheduler: policies to determine which process to execute next

Process dispatcher: actual binding of selected process to a processor (switching context, switching to user mode, jumping to the proper location in the program).

Decision mode: activating process scheduler

Dit document is beschikbaar op studeersnel

Downloaded by Anonymous Student (danieltyukov1@gmail.com)

Decision mode defines conditions for activating the process scheduler

Upon activation

- A number of processes in memory that are ready to execute
- Process scheduler selects one for execution

Scheduler can be invoked (decision mode) e.g. when a process:

1. Switches from running to waiting state
2. Switches from running to ready state
3. Switches from waiting to ready
4. Terminates

If scheduling takes place ONLY under 1 & 4 (active process voluntarily yields)

➔ Non-preemptive decision mode

Otherwise

➔ Preemptive decision mode.

First-come, first-served (FCFS) scheduling

Decision mode: non-preemptive

Prioritiy function: the earlier the arrival, the higher the priority.

Arbitration rule: random choice among processes that arrive at the same time.

Shortest job first (SJF) scheduling

Schedule the process with the shortest next CPU burst.

Decision mode: nonpreemptive

Priority function: equal to -t where t is the length of the next CPU burst

Arbitration rule: chronological or random ordering

SJF is optimal

- Gives minimum average waiting time for a given set of processes
- The difficulty is knowing the length of the next CPU burst

Priority scheduling

A priority number (integer) is associated with each process

- Explicitly assigned, by the programmer, or
- Computed through a function that depends on task properties, e.g. memory use, timing: duration (estimated), deadline, period

CPU is allocated to the process with highest priority

- Preemptive: priority_of_any_running_process >= priority_of_any_process_in_ready_queue

Downloaded by Anonymous Student (danieltyukov1@gmail.com)

- Non-preemptive is a word I know

Problem: starvation -> low priority processes may never execute

Solution: aging -> increase the priority of the process over time.

Round Robin (RR) scheduling

Each process gets a small unit of CPU time (time quantum of length q). At the end of a time quantum, the process is pre-empted (ANOTHER word I know!) and is added to the end of the ready queue

Decision mode: if a process runs q continuous time units, it is pre-empted

Priority function: like first come first serve (though not really used like that)

Arbitration rule: random ordering

If there are n processes in the ready queue, each process gets 1/n of CPU time (for small q) in chunks of at most q time units at once. At most (n-1)q time units until a process is given CPU time again.

Higher average turnaround than Shortest Job first, but better response.

Time quantum and context switch time.

Performance: q large => FCFS, q small => q must be large with respect to context switch time, otherwise context switch overhead is high.

Multilevel queueueueueueueueue scheduling

Ready queue is partitioned into various sub-queues

Each queueueueueueue has its own scheduling algorithm

- Decision mode: preemptive (RR) or non-preemptive (FCFS)

Also: scheduling between the queues

Option 1: fixed priority scheduling -> possibility of starvation

Option 2: time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes.

Multilevel feedback queue

Instead of fixed category (fixed sub-queue) for a process, a process can move across sub-queues; aging can be implemented this way

Scheduler is defined by the following parameters:

- Number of queues
- Scheduling algorithms for each queue
- Method used to determine when to upgrade a process
- Method used to determine when to demote a process

Dit document is beschikbaar op studeersnel

Downloaded by Anonymous Student (danieltyukov1@gmail.com)

- Method used to determine which queue a process will enter when that process needs service.

Rate monotonic (RM) scheduling

In real time systems, processes are typically periodic in nature. A process with period T is activated every T time units, and its computation must complete within a relative deadline of D time units.

Decision mode: preemptive.

Priority function: shorter T -> higher priority

Arbitration rule: chronologic or random ordering.

Earliest deadline first (EDF)

As in RM, real time system and periodic processes. Highest priority is assigned to the process with the smallest remaining time until its deadline

Decision mode: preemptive

Priority function: least time remaining until deadline -> higher priority

Arbitration rule: chronological or random ordering.

Priority inversion

A low priority task obtains a resource, a high priority task waits on it. A middle priority task pre-empts the low priority task. The high priority task now waits on the middle priority task and executes effectively at the low priority.

Solution: priority inheritance protocol. The priority of task P using the resource is dynamically adjusted to be the maximum of the priority of any other task that is blocked on the allocated resources of P and its own priority. Middle priority tasks will wait now.

Concurrency and atomicity

Interfering activities are inherent in OS and concurrent programs.

Sources: interrupt servicing, OS activities, multiple threads on same global data, arbitrarily switched, multiple processors, multiple active devices, sharing hardware resources (memory, bus)

The sequential process: discrete sequence of states. Needed: indivisible, atomic steps/actions. Execution never observed to be half-way an atomic action.

In processes, the finest level of detail wrt progress consists of atomic actions.

Single reference rule: a statement in a programming language may be regarded as atomic if at most one reference to a shared variable occurs.

Problem addressed by single reference rule

In between any pair of instructions of one process, (part of) another process or collection of processes can be executed, including the OS

Problem: old copies of shared variable can be stored in internal registers or in memory locations.

Downloaded by Anonymous Student (danieltyukov1@gmail.com)

Concurrent execution

Concurrent processes take a joint path through the joint state space time continuum

Trace: sequence of atomic actions, obtained by interleavings of concurrent parts. Maintains the internal execution order of the individual processes. There are many possible traces in the joint state space time continuum.

Traces

Traces are sequences of actions, they represent the steps that a program goes through. Actions are assignments or tests. A trace is a possible one if all its tests yield true. Being possible depends on initial program state. Traces can be finite or infinite, and a program text has many traces.

The traces of a concurrent program are obtained by interleaving traces of all concurrent parts.

It is impractical to determine all traces. However most often considering a limited number suffices. They can be generated automatically and be useful in gaining insight.

Synchronization refers to ordering (restricting possible paths or forbidding certain traces) typically through indicating event occurrences.

Peterson's algorithm

Combine the ideas for mutual exclusion between two processes:

- Take turns in crowded circumstances (use t)
- Don't wait if there is no need (look at bY)

Check correctness by detailed annotation. The annotation shows mutual exclusion and absence of deadlock. Use an auxiliary variable to record the state of the program in between the two statements. Show that no assignment in Py can disturb any assertion. Show that locally these assertions are indeed correct.

Proving facts: use program text and assertions. There are statements about all possible traces without the need to examine them explicitly.

Check exclusion, check deadlock.

Non-interference, formally

Assume that we have two processes and assignments with annotation based on local analysis. Then we must show that mutual disturbance is impossible: A0 does not disturb P1; A1 does not disturb P0.

- {P0 ^ P1}A0{P1}
- {P0 ^ P1}A1{P0}

Mutual exclusion

One way of looking at the bridge problem is to regard the bridge as a shared resource, acquired by cars from both directions, actions on the bridge must not be interleaved.

The bridge is accessed under mutual exclusioin, the entry/exit actions form a critical section.

Dit document is beschikbaar op studeersnel

Downloaded by Anonymous Student (danieltyukov1@gmail.com)

Programming mutual exclusion can be using the concept of a (binary) semaphore. In fact, turning sequences of atomic actions into a single one.

Semaphores (Dijkstra)

A semaphore is an integer s with intitititiitititititial value s_0 >= 0 and atomic operations P(s) and V(s). The effect of these operations is defined as follows:

P(s): <await(s>0); s: s-1 > -> proberen

V(s): <s:= s+1 > -> verhogen

"<>" denotes again defined atomicity, the implementation of P and V must guarantee this.

'await(s>0)' represents blocking until 's>0' holds. This is indivisibly combined with a decrement of s.

A semaphore is always non-negative.

Other names for P and V: wait/signal, wait/post, lock/unlock.

POSIX: mutex (1003.1c)

Special, two-state semaphore: mutex. Between threads, specifically for mutual exclusion.

Concepts in concurrency

Atomic action: finest grain of detail, indivisibly. Typically, assignments and tests in a program. Suficient at program level: single reference to shared variable in statement.

Parallel execution: interleaving of atomic actions.

Race condition: situation in which correctness depends on the execution order of concurrent activities.

Requirements on solutions

Functional correctness: satisfy the given specification. An assertion that is needed for correctness must not be disturbed.

Minimal waiting: waiting only when correctness is in danger

Absence of deadlock: don't manoeuvre (part of) the system into a state such that progress is no longer possible.

Absence of livelock: ensure convergence towards a decision in a synchronization protocol.

Fairness in competition: (weak) eventually, each contender should be admitted to proceed. (strong) we can put a bound on the waiting time of a contender. Absence of fairness: leads to starvation of processes.

Reasoning about parallel programs

Annotation: assertions at control points.

Assertion: predicate (Boolean function) in terms of the program variables.

Control point: place in the program text where one might inspect the current state.

Invariant: special assertion that holds at every control point.

Downloaded by Anonymous Student (danieltyukov1@gmail.com)

Action synchronization

Synchronization

Synchronization is about limitation of possible program traces by coordinating execution such that

- A certain invariant is satisfied
- Or the execution has some desired property

Typically, by blocking execution until an assertion has become true.

Await(A) denotes blocking until a condition A holds.

Issues around the example

Using repeated testing for await() works **if** assignments and tests are atomic. However, usually, there is a critical section which is not atomic and needs mutual exclusion. **And** at most 1 vendor and 1 machine exist, otherwise race conditions occur.

Repeated testing is called busy waiting and only acceptable when

- Waiting is guaranteed to be short or
- There is nothing else to do (e.g. in dedicated hardware)

Instead, rely on OS primitives for waiting.

Action synchronization

Action synchronization relies on action counting and invariants on the counting.

An invariant *I* is an assertion that holds at all control points.

Terminology: naming and counting

Naming of actions: if *A* is an action in the program, c*A* denotes the number of completed executions of *A*. c*A* can be regarded as an auxiliary variable that is initially 0 and is incremented atomically each time A is executed.

Topology properties

Topology invariants: derived directly from the program text

Example: two actions always occurring one after the other

Initially: x=0 ^ y=0

While true do A: <x: = x+1>; B: <y:=y+1> od

While true do C: <y:=y-1>; D: <x:=x-1> od

Invariants:

I_0: x = cA – cD

I_1: y = cB – cC

I_2: 0 <= cA- cB <= 1

I_3: 0 <=cC-cD<=1

Dit document is beschikbaar op studeersnel

Downloaded by Anonymous Student (danieltyukov1@gmail.com)

We can prove that I: y<= x holds using topology invariants.

y <= x = cB – cC <= cA – cD {I0 and I1}

= true {I2 and I3}

Synchronization conditions

Action synchronization is specified by inequalities on action counts or on program variables directly related to this counting.

We refer to such an inequality as a synchronization condition or a synchronization invariant.

Recall semaphores (Dijkstra)

Non-negative integer s with initial value s_0 and atomic operations P(s) and V(s) (try and increment). Semaphores can be used to implement mutual exclusion.

Semaphore invariants

From the definition, we derive two semaphore properties (invariants):

S0: s>= 0

S1: $s = s_0 + cV(s) – cP(s)$

S0, S1: functional properties (“safety”). Combining the two:

S2: $cP(s) <= s_0 + cV(s)$

Hence, semaphores realize a synchronization invariant by definition. Blocking is allowed only if the safety properties would be violated.

Action synchronization solution (in general)

Given: - collection of tasks/threads executing actions A, B, C, D;

- A required synchronization condition (invariant)

  SYNC: a *cA + c*cC <= b*cB + d*cD + e for non-negative constants a,b,c,d,e

Solution: introduce semaphore s, s_0 = e, and replace

A -> $P(s)^a$; A B-> B;$V(s)^b$

C-> $P(s)^c$;C D-> D;$V(s)^d$

Note: during execution of A and C we have strict inequality in SYNC.

Remarks: action synchronization

- One semaphore for each synchronization condition
- Synchronization conditions may be conflicting. A deadlock may result.
- Sometimes a deadlock can be avoided by imposing extra restrictions.
- Finding synchronization conditions can be painful.

Mutual exclusion

Downloaded by Anonymous Student (danieltyukov1@gmail.com)

Cooperation is not the only problem. Competition for resources (sharing) must be handled too. Given are N different processes, repeatedly executing a critical section. Maintain as synchronization requirement (for mutual exclusion)

M: the sum of the entries – the exits of the critical sections <= 1. (only one process in a critical section at a time)

Verifying correctness

Functional correctness and minimal waiting are by construction

Fairness: the solution is just as fair as the semaphore(s). depends on semaphore implementation and order of release from the waiting queue.

Deadlock:

A deadlocked state is a system state in which a set of threads or processes are blocked indefinitely. Typically, each thread is blocked on another thread in the same set.

Prove absence of deadlock, typically by contradiction. Assume a deadlock occurs, investigate which blocked sets are possible, show a contradiction (examine all possible combinations of blocking actions in all tasks.

Producer-consumer problem using bounded buffer

1. Order of values received equals order of values send
2. No receive before send
3. Number of sends cannot exceed number of receives by more than a given positive constant N (buffer size)

First requirement: data structure supporting FIFO

➔ Queue q, with operations PUT(q,x) and GET(q,y). Introduce variable q of type queue.

Exclusive access is required since PUT and GET are not atomic. Introduce semaphore m, $m_0 = 1$.

Second requirement translates into cGET(q,..) <= cPUT(q,…). Introduce semaphore t, $t_0=0$.

Third requirement translates into cPUT(q,…) <= cGET(q,…) + N. Introduce semaphore s, $s_0 = N$.

Optimization of code

Practice: use a finite array of length N (with indices modulo N)

Is it possible to leave out semaphore m for synchronization? Then the array may never be accessed at the same place. Semaphore m for exclusion is not needed! An array of size N, used in circular manner suffices???????

Preventing deadlock

Let critical sections terminate. No P operations between P(m) and V(m).

Use a fixed order in P-operations on semaphores.

Beware of greedy customers.

In general: avoid cyclic waiting!

Dit document is beschikbaar op studeersnel

Downloaded by Anonymous Student (danieltyukov1@gmail.com)

POSIX examples

Naming and creation:

Name within kernel, persistent until re-boot, like a filename. Also unnamed semaphores, for use in shared memory between processes. Hence, two interfaces for creation and destruction.

Condition Synchronization

Motivation

Action or event synchronization: the invariant refers to ordering of actions. Counting actions is enough. BUT just counting does not solve all synchronization problems. E.g. x: =x +y or wait until x=0 OR y=0 (disjunctions)

Condition synchronization: the invariant refers to a certain (combined) state. Needs explicit communication (signalling) between processes. Communicating via shared variables is not good enough! Needed when just counting is not enough to solve a synchronization problem. Simplifies sequences of semaphore operations.

Two condition synchronization principles:

At places where the truth of a condition is required: check and block.

At places where a condition may have become true: signal waiters.

Condition variables

Var cv: Condition; (Boolean function B defined on cv)

4 basic operations on variable cv:

- Wait(…,cv) : suspend execution of caller
- Signal(cv) : free ONE process/thread suspended on Wait(cv) (void if there is none)
- Sigall(cv) : free ALL processes suspended on Wait(cv)
- Empty(cv) : “there is no process suspended on Wait(cv)” (returns true or false)

There are many extra operations in specific implementations.

Combine: condition variable and semaphore

Need to combine “V(m); Wait(cv)” atomically: define Wait(m,cv): <V(m); Wait(cv)>; P(m)

Program structure

Structure program in condition critical regions.

Check the truth of condition before executing the critical section. Use repetition rather than a selection, to be safe. Take competition into account.

Decide carefully which variables must be protected together. Usually, these occur together in at least one condition. Access to these variables only within critical sections. Reads in other parts are harmless, but unstable.

Downloaded by Anonymous Student (danieltyukov1@gmail.com)

A typical region looks like:

- Mutex lock
- While not condition do wait od;
- Critical section
- Possible signals
- Mutex unlock

Using condition variables

Signalling and waiting must be in critical sections. Needs an associated exclusion mechanism. *cv* is sometimes extended with a timeout mechanism, recheck even though there is not signal.

Each condition variable cv is associated with a condition B(cv). Signalling means B(cv) may have become true. B(cv) can be a simple condition or a combined condition. Alternatively, multiple condition variables can be used, leading to a single semaphore associated with all critical sections. Usually, a single condition variable is used for all conditions. Hence, using more than one is an efficiency choice.

Rule: signal when a condition might have become true.

Signalling strategies

The only problem remaining is the choice of signalling type.

Two signalling strategies:

- Signal one waiter of cv satisfying: “B(cv) is true and there are waiters on it”-> Empty(cv) = false
- Sigall all waiters of the condition variable (of which the associated condition may have changed from false to true)

Which one to choose: Signal or Sigall. Careful analysis leads to efficiency. Sigall is always correct, but inefficient.

Monitors.

Definition: a monitor is combination of data and operations (procedures, methods) on data.

In fact: an object, like in object-oriented languages.

Procedures of a monitor are executed under mutual exclusion. Procedure executing is said to be “inside the monitor” (only 1 can be inside) (like a woman)

Synchronization is through condition variables.

The effect on the caller of a signal depends on the monitor implementation. Some implementations give up the monitor immediately. This signalling semantics is called signalling discipline.

Communication is through the local variables of the monitor.

Structuring by monitors

Dit document is beschikbaar op studeersnel

Downloaded by Anonymous Student (danieltyukov1@gmail.com)

Condition variables are rather useless without combining with proper mutual exclusion. Monitors provide the exclusion. At most one thread may be inside the monitor at any given time.

Programs with monitors employ:

- Active processes, passive monitors
- Relevant shared variables that are encapsulated
- Well-defined operations on shared variables

Scheduling/ Signalling disciplines

Four disciplines are used in various implementations

- Signal and exit: the process executing a signal exits the monitor immediately. Monitor access is given to a waiter on that variable, if any. Sigall is not supported. Signalling strategy: upon leaving the monitor, a process performs at most one signal on a condition variable that is non-empty, and for which the corresponding condition holds. Property: the condition holds after wait.
- Signal and continue: the process executing a signal continues to use the monitor, until a wait or until the end of the routine. Any processes released by this signal compete with the other processes to obtain monitor access again. Signalling strategy: during execution of a monitor routine all relevant conditions are signalled. After each wait, the condition is evaluated again. Property: the condition may or may not hold after wait.
- Signal and (urgent) wait: the process executing a signal is stopped in favour of the signalled process. The signaller waits to access the monitor again after the Signal, and competes with all other processes again. Urgent wait: signaller has priority over other users. Signalling strategy: similar to signal and exit. Signalling is done upon leaving the monitor. Property: the condition holds after wait.

General strategies

Use a repetition to evaluate and wait on a guard. Most disciplines have unpredictability in the choice of the next process in the monitor.

Upon each monitor exit,

SIGNAL: employ cascaded wakeup by using just Signal(cv) : if B(cv) holds and cv is not empty,

SIGALL: wake up the entire group waiting on each variable cv for which B(cv) may have become true.

Signal at the end of a routine.

Do not change the state before waiting. Beware that monitor entry and exits include the Wait(). By not modifying the state, signalling is not needed before the Wait().

Two ways to determine which variable to signal:

- just signal (resp sigall in 2nd strategy) each cv with a true guard.
- Evaluate the status of each cv and don't signal more than necessary

In 'sigall strategy' several conditions per cv can be used, the sigall must then be performed if any of these conditions may have become true

Downloaded by Anonymous Student (danieltyukov1@gmail.com)

Deadlock

Starvation: can be the result of cooperation of several processes. Can also be the result of interference between scheduling and blocking.

Livelock: try to pass a critical condition repeatedly, without making progress. Results functionally in starvation ordeadlock.

Deadlock: extreme case of starvation: continuation not possible.

Terminology

Deadlock is usually associated with access to resources.

Consumable resources: resource use takes it away (variable number)

- Typical producer/consumer problems

Reusable resources: resource is shared (fixed number)

- Typical mutual exclusion problems (critical section), readers/writers type of problems.

Formal definitions

We call a task (process or thread) blocked if it is waiting on a blocking synchronization action or has terminated.

A set D of tasks (with at least one non-terminated task) is called deadlocked if all tasks in D are blocked, and for each non-terminated task t in D, any task that might unblock t is also in D.

Conditions

Program behaviours that may lead to deadlock:

- Mutual exclusion
- Greediness: hold and wait
- Absence of pre-emption mechanism
- Circular waiting

These all play a role and should be addressed explicitly in the solution.

Model for analysis: graphs

For consumable resources and general condition synchronization, the graph representation is the wait-for graph. Nodes are tasks, edges area wait-for (blocked-on) relationship.

Wait-for graph

The graph captures a possible dynamic situation, a system state. Possibility of existence of the graph needs evidence. May label the arrows with corresponding blocking conditions.

Note: we leave out the dependency on m since we know mutual exclusion does not add to deadlock provided that the critical sections terminate.

More on graphs

Dit document is beschikbaar op studeersnel

Downloaded by Anonymous Student (danieltyukov1@gmail.com)

With reusable resources and action synchronization, we use the resource dependency graph. This is a bipartite graph with two classes of nodes: tasks and resources. There are three types of edges, capturing states and classes of states:

- Type1: task has requested and now waits for the resource
- Type2: task has acquired (holds) the resource
- Type3: task may request the resource

Each resource graph represents a particular state of the system.

Three events change the state:

- Request (by a task)
- Acquire (response to a request by the system, according to a policy)
- Release (by the task)

Resource dependency graph

P is blocked if it has an outgoing arrow that is not directly removable. Directly removable: the requested resource is free.

Analysis: reduction

Assume that the graph represents a stable state, repeatedly remove a non-blocked task and all its incoming connections. This represents possible completion of that task. Remaining set: deadlocked. A knot.

Sufficient condition for deadlock for the greedy allocation policy: existence of a knot in the original graph. Greedy: direct allocation decision based on availability. This greedy allocation could be implemented just with a semaphore (counting the resources)

Dealing with deadlock

Deadlock is often ignored (since rare). Timeouts, external intervention.

Approaches:

- Prevention
- Avoidance (system side)
- Detection and recovery (possible roll-back).

Prevention

Use reduced dependency graphs and wait-for graphs. Make the reduced dependency graphs empty by construction, and avoid cycles in the wait-for graphs.

Use synchronization tricks. Prevent circular wait, have terminating critical sections. This is not always possible.

Use pre-emption of the resource when needed. This is not always possible.

Acquire “all resources at once”: avoid the “wait-and-hold” greediness. Leads to possible starvation.

Downloaded by Anonymous Student (danieltyukov1@gmail.com)

Prove correctness

Typically by contradiction. Assume a deadlock occurs, show a contradiction.

Examine all possible combinations of blocking states in all tasks! Examine the corresponding dependency and wait-for graphs and show that deadlocked ones are not possible or not reachable.

Avoidance: prevent from system side

Maintain as a system invariant that no deadlocked sets can occur. Upon each blocking action: investigate if always an open execution path remains (system remains in a “safe” state). Otherwise, deny or postpone the action. Also, just make sure the reduced dependency graph is empty!

Postponing works if the blocking actions refer to resource allocations and we can compute the future states. Example: Banker’s algorithm needs information about possible system behaviours in terms of resource requirements (maximum numbers).

Banker’s algorithm: problem description

Given: set of N tasks, set of R resources. C[j] : number of type j resources, max[I,j] : maximum number of type j resources needed by task i.

Tasks acquire resources incrementally and release those eventually. Requirement: synchronize requests such that always each task can acquire resources until its specified maximum. Not satisfying this requirement represents a deadlock.

(maximum) claim graph

The (maximum) resource claim is illustrated in the resource dependency graph, using (dashed) arrows. The maximum claim graph.

Problem analysis

Problem: while giving out resources arbitrarily, a state may be reached such that no additional request can be served of some tasks (not enough left), ever. Hence these tasks can never proceed, and will also never give back their reservations. Actually, an instance of wait-and-hold.

Formalization

A state is called safe for a task if this task can be given its maximum number of resources eventually. Possibly by giving available resources to other tasks first and then waiting until these release them again. If that is not needed the state is called open for that task

A state is called safe in general if it is safe for all tasks.

The initial state must be safe.

A new state resulting from granting a request is accepted only if it is safe. Just requesting does not change state safety. Now the problem reduces to verification of state safety.

State safety upon new request

Assume the current state is safe

Consider a new request by task I (req[j] denotes the number of type j resources requested)

The new state (if resource is allocated) is obtained as follows:

Dit document is beschikbaar op studeersnel

Downloaded by Anonymous Student (danieltyukov1@gmail.com)

-av, alloc[i], claim[i] : = av-req, alloc[i]+req, claim[i]-req

It is enough to verify whether this new state is safe for just task i. if it is, then eventually all its resources will be returned, thus reaching an even safer state than the one before the adjustment.

Task I is safe when its claims can be satisfied, either directly or by completing any of the other tasks (this must be verified)

Solution: repeatedly

- Find an open task
- Remove it, and all its claims (graph reduction)
- Until task I is found to be open

Detection

Invoke detection algorithm periodically to check if deadlock occurs. An algorithm to examine the state upon a blocking action. Also an algorithm to recover from deadlock. Roll-back may be necessary.

Repeatedly monitoring the system. Gives overhead and needs detection algorithm.

Dealing with deadlock detection

- Locally, inside the task that tried blocking
- Globally, through a recovery policy:

➔ Kill : all or selectively, based on criteria

➔ Roll back to safe state: works only if alternatives exist, need recording checkpoints

➔ Pre-empt resources, if possible. Select victim based on criteria. However, one may argue that the deadlock does not exist in this case.

Downloaded by Anonymous Student (danieltyukov1@gmail.com)

Dit document is beschikbaar op studeersnel

Downloaded by Anonymous Student (danieltyukov1@gmail.com)